Esperto in un click

Francesco Gatto Sara Bosco

Windows Phone 8

Corso di programmazione pratico

LIVELLO 1 **2** 3 4 5 6 7 8 9 10 11 12 13 14 15

Approfondisci l'uso dell'interfaccia Metro e crea un'app semaforo

IMPARERAI:

- a impiegare gli oggetti e le classi in C#
- a utilizzare i passaggi di parametri
- a realizzare un'app semaforo
- a sfruttare le potenzialità dei controlli Panorama e Pivot

area51 Publishing
www.area51publishing.com

Francesco Gatto e Sara Bosco

Windows Phone 8

Corso di programmazione pratico

Livello 2 - *Approfondisci l'uso dell'interfaccia Metro e crea un'app semaforo*

area51 Publishing

© 2014 Area 51 s.r.l., San Lazzaro di Savena (Bologna)

Prima edizione ebook Area51 Publishing: gennaio 2014

Windows Phone è un marchio registrato di Microsoft Corporation

Cover: Valerio Monego

Redazione e sviluppo ebook: Enrico De Benedictis

Se intendi condividere questo ebook con un'altra persona, ti chiediamo cortesemente di scaricare una copia a pagamento per ciascuna delle persone a cui lo vuoi destinare. Se stai leggendo questo ebook e non lo hai acquistato, ti chiediamo, se ti piace, di acquistarne anche una copia a pagamento, al fine di poterci permettere di far crescere il nostro lavoro e di offrirti sempre più titoli e una qualità sempre maggiore. Grazie per il tuo aiuto e per aver rispettato il lavoro dell'autore e dell'editore di questo libro.

Data la rapidità con cui i tool di sviluppo e i linguaggi vengono aggiornati, i contenuti di questo ebook si intendono fedeli allo stato dell'arte al momento della pubblicazione.

Se stai utilizzando un tablet (iPad, Android, Windows 8 etc.) per consultare questo ebook, ti ricordo che con un doppio tap (iPad) o con la funzione Zoom sull'immagine otterrai l'ingrandimento a tutto schermo.

ISBN:

Segui Area51 Publishing :

Google Plus

Facebook

Twitter

Pinterest

www.area51editore.com

www.area51editore.com/blog

Introduzione alla collana

Windows Phone 8 è l'innovativo sistema operativo mobile, realizzato da Microsoft, che si presenta con una grafica tutta nuova e una struttura di base totalmente rivoluzionata rispetto alle precedenti versioni di Windows Phone.

Semplice ed efficace, la programmazione su WP8 è consigliata a programmatori esperti e ai neofiti che faranno di questi ebook la loro prima esperienza da sviluppatori.

La collana è strutturata in modo tale da fornire le conoscenze di base del linguaggio di programmazione C# negli ambienti dedicati a Windows Phone 8, trovando immediato riscontro pratico in una o più applicazioni realizzate a fine lezione. A differenza di un comune manuale, ogni ebook spiega passo passo come realizzare applicazioni ben

strutturate e di successo sull'ambiente di sviluppo Visual Studio. Ogni pubblicazione risulta indipendente dalle altre (successive o precedenti), infatti sarà costituita da un'utile introduzione, una parte teorica con pochi concetti ben esplicati cui seguirà la realizzazione di applicazioni per consolidarli e approfondirli. Tuttavia, è presente una propedeuticità intrinseca: ogni lavoro è più complesso rispetto al precedente e dà per acquisiti i vecchi concetti evitando di annoiare il lettore con continue ripetizioni e fornendo uno strumento di apprendimento davvero immediato.

La collana è destinata ai potenziali sviluppatori per Windows Phone 8 che si approcciano per la prima volta alla programmazione; agli appassionati del settore che cercano un valido approfondimento e, perché no, un piacevole passatempo; a chi è già dentro il mondo della programmazione ma non conosce sufficientemente C#. Le applicazioni realizzate saranno complete e

correlate alla teoria, ma sapranno dare un valido spunto per progetti tutti nuovi e, a fine raccolta, ogni lettore sarà in grado di realizzare ciò che più gli piace proprio grazie agli strumenti appresi dai nostri esercizi predefiniti.

Parte 1

La programmazione

1. Stile di programmazione

1.1 La programmazione orientata agli oggetti

La programmazione orientata agli oggetti (OOP, Object Oriented Programming) rappresenta uno stile fondamentale di programmazione.

Essa comporta il raggruppamento di strutture dati ed elementi procedurali in classi (parti di codice sorgente che definiscono i tipi di dato). La diffusione e il successo della OOP sono dovute a un livello di astrazione superiore rispetto alla programmazione procedurale, al codice molto più facile da gestire e alla riusabilità.

Annoveriamo tra i più noti linguaggi orientati agli oggetti Java, Visual Basic, Delphi, C++ e ovviamente il nostro C#.

2. Oggetti e classi in C#

2.1 Classi e oggetti

Le classi sono frammenti di codice sorgente deputati alla definizione di nuovi tipi di dato. Questi ultimi saranno identificati tramite caratteristiche e proprietà che astraggano in modo semplice ed efficace uno o più oggetti reali. Le classi definiscono un insieme di oggetti in base ai loro connotati e non per elenco degli oggetti stessi. In C# quindi, le classi specificano come gli oggetti sono strutturati e come essi si comportano. L'oggetto, infatti, altro non è che un'istanza di una determinata classe.

L'istanza rappresenta un elemento distinto, allocato e configurato in memoria in maniera concreta e che può assumere valori propri. Finché un'area di memoria non è allocata, nessun oggetto può esistere. Ogni istanza è separata dalle altre, ma

presenta le stesse caratteristiche generali degli altri oggetti appartenenti alla stessa classe.

Se la classe è un'idea l'istanza è la sua realizzazione: si pensi al concetto di "Penna", il colore dell'inchiostro, il materiale e la forma di essa sono caratteristiche comuni a tutte le penne, ma ognuna di esse non le presenta necessariamente uguali a quelle di un'altra (penne blu e penne rosse, ad esempio, sono caratterizzate dal colore dell'inchiostro ma è diverso nelle une e nelle altre). Da qui si può dire che la "classe penna" costituisce lo stampo di ogni "oggetto penna", e che ogni istanza è un elemento presente in memoria che può assumere svariati valori.

2.2 Definire una classe

Definiamo una classe con i seguenti codici:

```
[modificatori] class Nome
{
```

```
        ...
}
```

Nel caso della classe “Punto” avremo ad esempio:

```
public class Punto
{
        ...
}
```

Il campo [modificatori] è “public”, un modificatore che estende la visibilità di questa classe a tutte le altre. In C# la visibilità della classe può essere estesa, tramite gli *access specifier* (specificatori di accesso o descrittori di visibilità), a quattro livelli: classe, sottoclasse, mondo esterno e package (un insieme di classi e interfacce che intervengono nello stesso contesto). Più in là utilizzeremo anche “internal”, un modificatore che limita la visibilità della classe ai soli elementi appartenenti allo stesso assembly (un insieme di funzionalità che vengono distribuite in una sola

implementazione) della classe.

2.3 Variabili e metodi

All'interno di una classe distinguiamo variabili e metodi.

2.3.1 *Variabili*

Le variabili (o variabili di istanza) attribuiscono a una classe le caratteristiche che dovrà presentare. Queste possono comprendere tipi primitivi quali int, double, bool, char, ma anche altri oggetti. La sintassi per dichiarare una variabile è la seguente:

```
[modificatori] tipo Nome
```

I modificatori delle classi sono quattro:

- `public` la variabile è visibile in tutte le altre classi;
- `private` limita la visibilità solo alla classe stessa;
- `protected` la visibilità è limitata alla classe e alle sue ereditarie

(l'ereditarietà è un legame stabilito tra due tipi);

- `internal` dichiara la visibilità della classe solo all'interno del package corrente.

Se si volesse creare una classe "Punto" nel piano cartesiano, la classe in questione necessiterebbe di due coordinate e, quindi, di due variabili di tipo reale:

```
private double X;
private double Y;
```

In questo modo si è creato "lo stampo" per ogni oggetto Punto, che avrà queste due variabili come "caratteristica".

2.3.2 Metodi

I metodi rappresentano essenzialmente un blocco di codice che consenta di effettuare operazioni sui tipi:

```
[modificatori] tipoRitorno
```

```
Nome (parametri)
{
        ...
}
```

I modificatori sono identici a quelli delle variabili e mantengono le stesse funzioni. Il tipo di ritorno identifica il risultato che l'elaborazione del metodo restituisce. Nel caso l'elaborazione sia passiva, e quindi non produca output, il tipo di ritorno sarà "void" (nulla). Il metodo può ricevere delle variabili che completino l'elaborazione, queste sono dette parametri e sono opzionali. Ogni metodo che restituisce un tipo (quindi non void) deve includere al suo interno l'istruzione "return". Questa servirà a mandare in output il frutto di un'elaborazione.

Ecco un esempio:

```
public int Somma(int a, int b)
{
        int Somma = a + b;
```

```
        return Somma;
}
```

Esiste un metodo particolare chiamato “Costruttore” che permette di definire le modalità di creazione di un oggetto, e quindi, come esso debba essere istanziato. Qui è bene scrivere tutto il codice che definirà i valori iniziali delle variabili d’istanza. La sua definizione è identica a quella di un qualsiasi metodo, ma differisce per l’assenza del nome e per il tipo di ritorno che è la classe di appartenenza.

Un esempio completo sulla classe “Punto”:

```
public class Punto
{
        private double X;
        private double Y;

        public Punto()
        {
            X = 0.0;
```

```
        Y = 0.0;
    }
}
```

2.4 Dichiarare un oggetto

Un oggetto può essere considerato come una variabile qualunque e può essere dichiarato anche senza inizializzazione:

```
TipoOggetto nome;
```

Molto spesso, però, si utilizza l'inizializzazione diretta:

```
TipoOggetto nome = new
TipoOggetto ([parametri]);
```

L'istruzione "new" corrisponde all'allocazione diretta di un oggetto in memoria ed è necessaria per creare un'istanza.

È possibile accedere alle variabili di un oggetto o invocare il metodo di tale oggetto tramite la seguente sintassi:

```
nomeOggetto.NomeVariabile
nomeOggetto.NomeMetodo([parametri]);
```

La visibilità dei metodi e delle variabili deve essere estesa alla classe in cui operiamo. Qualora lavorassimo su un'altra classe non sarebbe, ad esempio, possibile richiamare l'accesso alla variabile (private) "X" di un'istanza dell'oggetto "Punto".

3. Passaggio di parametri in C#

3.1 Argomento dei metodi

Quando una funzione agisce su degli argomenti, essa può utilizzarli per giungere ad un risultato o modificarli.

I metodi vengono dichiarati all'interno di una classe specificando, oltre al nome, il livello di accesso, il valore restituito e, se richiesti, i parametri del metodo. I parametri sono racchiusi tra parentesi ( ) e separati da virgole. È possibile lasciar vuote le parentesi qualora non siano necessari parametri dei metodi.

In C# è possibile avvalersi del passaggio di tali parametri per valore o per riferimento garantendo quindi (proprio tramite il passaggio) una linea di comunicazione diretta tra la funzione chiamante e quella chiamata.

3.2 Passaggio di parametri per valore

Il modo più semplice di passare un dato ad una funzione è il passaggio per valore. Tale passaggio garantisce alla funzione l'accesso al parametro in questione, ma non gli consente di apportare ad esso modifiche definitive.

Ecco un esempio di passaggio per valore:

```
int x = 5;

x++;

Resetta(x);

x++;

…

public void Resetta (int i) {

     i=0;

}
```

La variabile intera x inizialmente vale

5; dopo un primo incremento, il suo valore diventa 6. A questo punto viene chiamato il metodo Resetta sul valore di x=6. L'intero i, nella funzione, contiene una copia del valore di x, in questo caso quindi vale 6. Dopo l'istruzione i=0, il valore di i diventa 0 ma quello di x resta 6 (invariato). Nella terza riga viene chiamata nuovamente l'istruzione x++ che rende x = 7 (e non 1 perché non è stata x ad essere resettata a zero, ma una sua copia chiamata i). Il passaggio per valore opera tramite una copia che non modifica il valore originario. È utile affidarsi a questo metodo qualora non si voglia modificare il valore del parametro, ma lo si debba utilizzare per elaborare un altro risultato, ad esempio per restituire il valore della radice quadrata di x senza modificare x stessa.

3.3 Passaggio di parametri per riferimento

Quando anziché un tipo di base venga

passato un oggetto come parametro, interviene il passaggio per riferimento. Il metodo riceve un riferimento all'oggetto e non una sua copia, in modo tale che i cambiamenti apportati all'oggetto si ripercuotano anche fuori dal metodo. È possibile ottenere questo risultato anche utilizzando le parole chiave *ref* e *out*.

```
int x = 5;

x++;

Resetta(ref x);

x++;


…


public void Resetta (ref int
i) {

     i=0;

}
```

Il valore finale di x sarà realmente 1 perché i cambiamenti apportati all'interno del metodo Resetta sono permanenti.

Ref è stato utilizzato ipotizzando che

il valore della variabile sia già definito. Quando si adopera la parola chiave *out* si vuole che la variabile, inizialmente senza valore, venga inizializzata dalla funzione stessa.

Parte 2

IL SEMAFORO:

IMMAGINI 2D E NAVIGAZIONE

TRA LE PAGINE

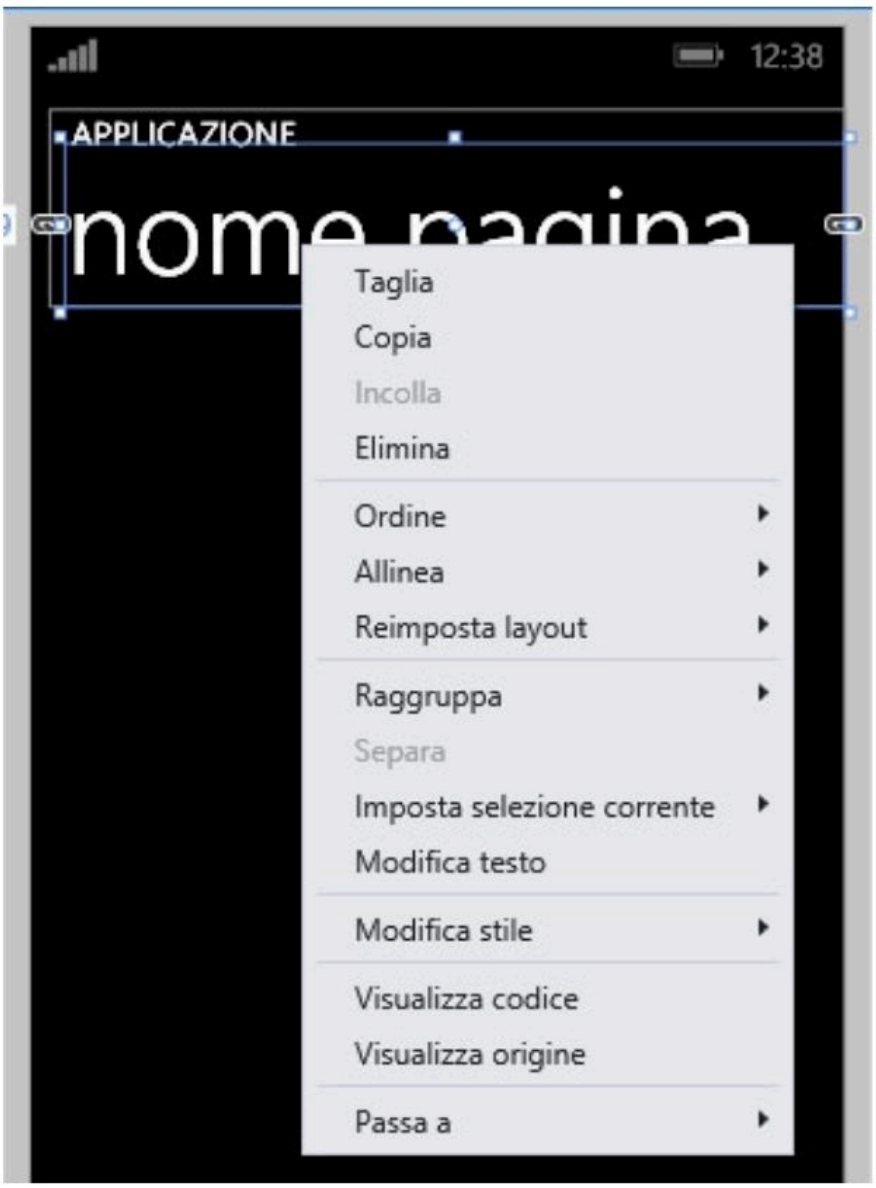
4. Elementi pratici per le app

Grazie al livello 1 di questo corso di programmazione, abbiamo appreso le nozioni necessarie alla creazione di utili elementi per le applicazioni Windows Phone. In questo secondo livello inizieremo ad introdurre nuovi oggetti (con relativa implementazione nel progetto) e lo studio di app dinamiche tramite le modifiche dell'interfaccia grafica. Grazie agli esempi qui descritti sarà possibile anche creare progetti su più pagine e sfruttare le variabili globali.

5. Creare e gestire un nuovo oggetto: il semaforo

5.1 Creazione dell'interfaccia grafica

Dopo aver creato un nuovo progetto, basterà impostare il modello "Applicazione Windows Phone" e rinominare la soluzione "Semaforo". Selezioniamo la "TextBlock" con testo "nome pagina", clicchiamo col tasto destro del mouse e selezioniamo "Elimina" come in figura.

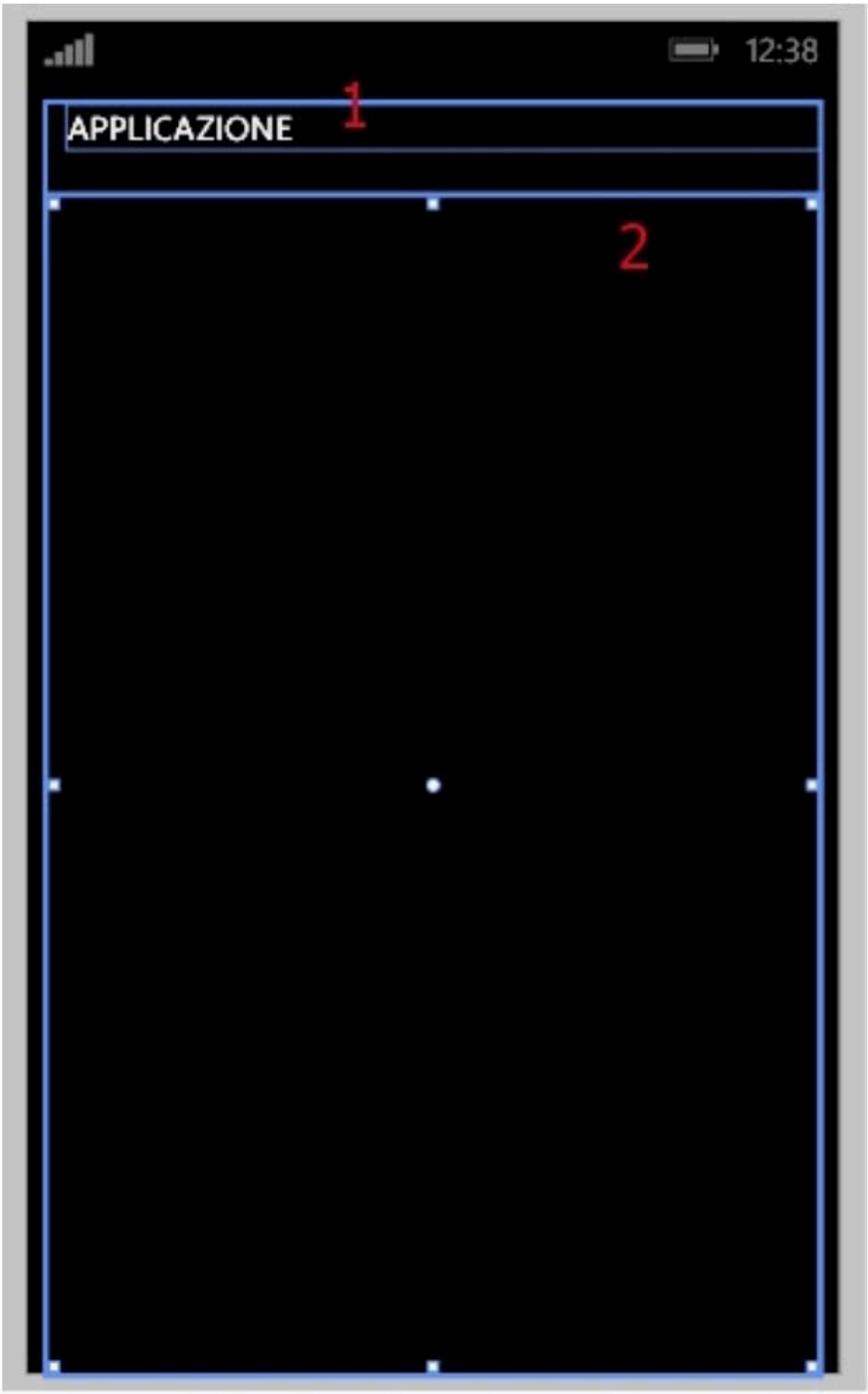


L'emulatore si presenta diviso in due parti.

1. In alto il "Textblock" applicazione.
2. Nel corpo della pagina una struttura chiamata "Grid", delimitata da un rettangolo. Questo elemento rappresenta una

griglia su cui è possibile formattare gli elementi appoggiandosi alla struttura definita da “Row” (riga) e “Columns” (colonne).

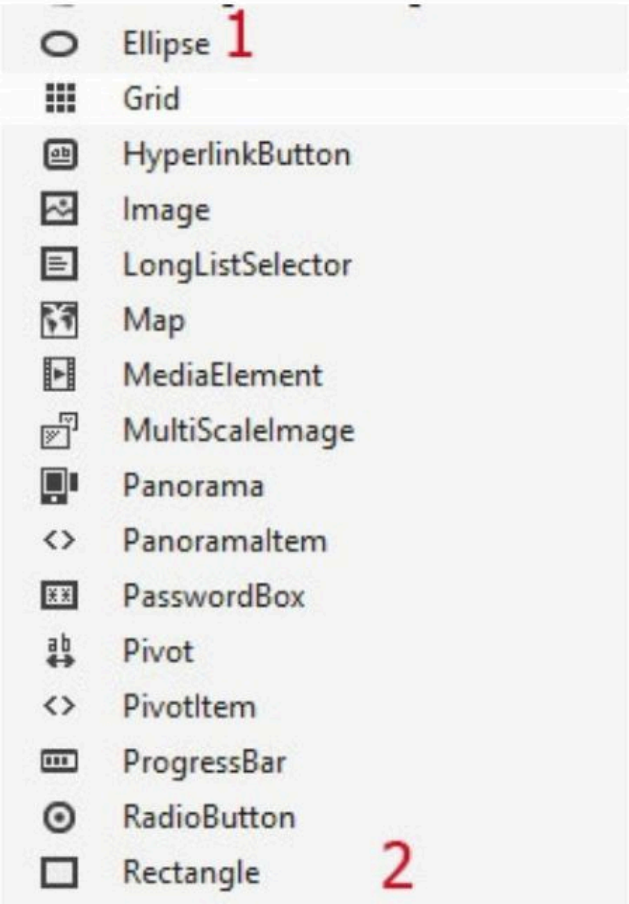
Per allungare l’elemento Grid trasciniamo il bordo superiore allargandolo quasi fino a toccare il TextBlock soprastante.



5.2 Introduzione di figure 2D

Una volta preparata l'area di lavoro, apriamo la casella degli Strumenti. In questa prima applicazione ci torneranno utili "Ellipse" e

“Rettangle” che definiscono rispettivamente gli elementi grafici “ovale” e “rettangolo”.

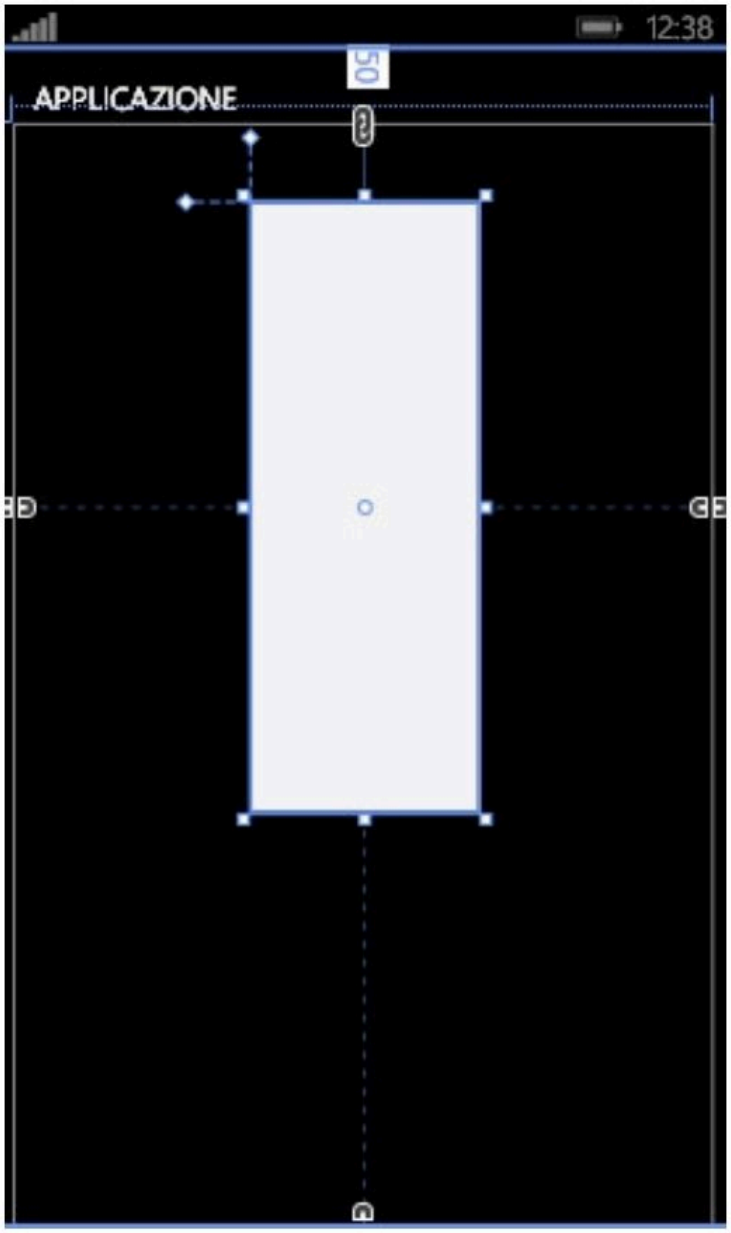


L’oggetto (il rettangolo in questo caso), una volta selezionato, va trascinato sull’emulatore. In basso a destra andremo a modificare i parametri associati utilizzando i seguenti valori:

- Height = 400;
- HorizontalAlignment = Center;
- Margin = 0, 50, 0 ,0;

- VerticalAlignment = Top;
- Width = 150;

Infine modifichiamo l'attributo "Fill" impostando un colore simile a un violetto chiaro. Alla fine dovremmo avere una situzione simile a quella in figura:



Il rettangolo appena definito sarà la scatola del semaforo. Ora procediamo con la creazione delle luci al suo interno. Allo stesso modo del rettangolo, trasciniamo l'oggetto Ellipse sull'emulatore. Sarà utile ripetere l'operazione fino a inserire 3 oggetti e, per ognuno di essi, imposteremo le proprietà come segue:

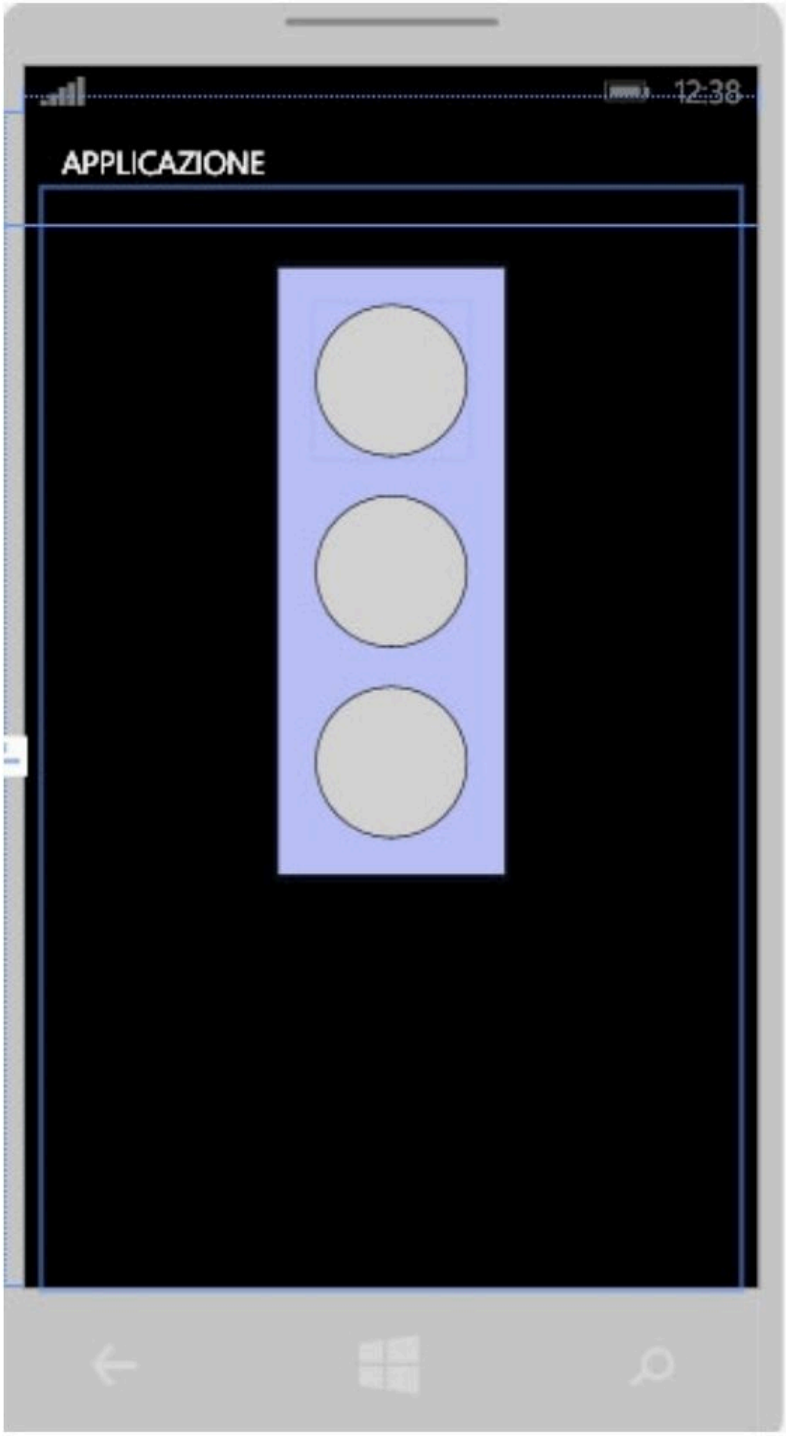
1. Height = 100;
2. HorizontalAlignment = Center;
3. VerticalAlignment = Top;
4. Width = 100;
5. Fill = LightGray (grigio chiaro);

Le proprietà appena definite sono comuni a tutte e tre gli oggetti ma, com'è facile intuire, sarà necessario definire i margini in modo differente:

- ellipse1 : Margin = 0, 75, 0 ,0;
- ellipse2 : Margin = 0, 200, 0 ,0;
- ellipse3 : Margin = 0, 325, 0 ,0;

Abbiamo così ottenuto le forme del

semaforo:



Per completare l'interfaccia, mancano solo i bottoni di controllo. I bottoni saranno quattro, di cui tre per i rispettivi colori del semaforo e uno per la stampa di una notifica.

Inseriamo i quattro bottoni nell'emulatore (se non ricordate come, controllate il primo volume del corso) e per ognuno di essi definiamo le proprietà.

button1:

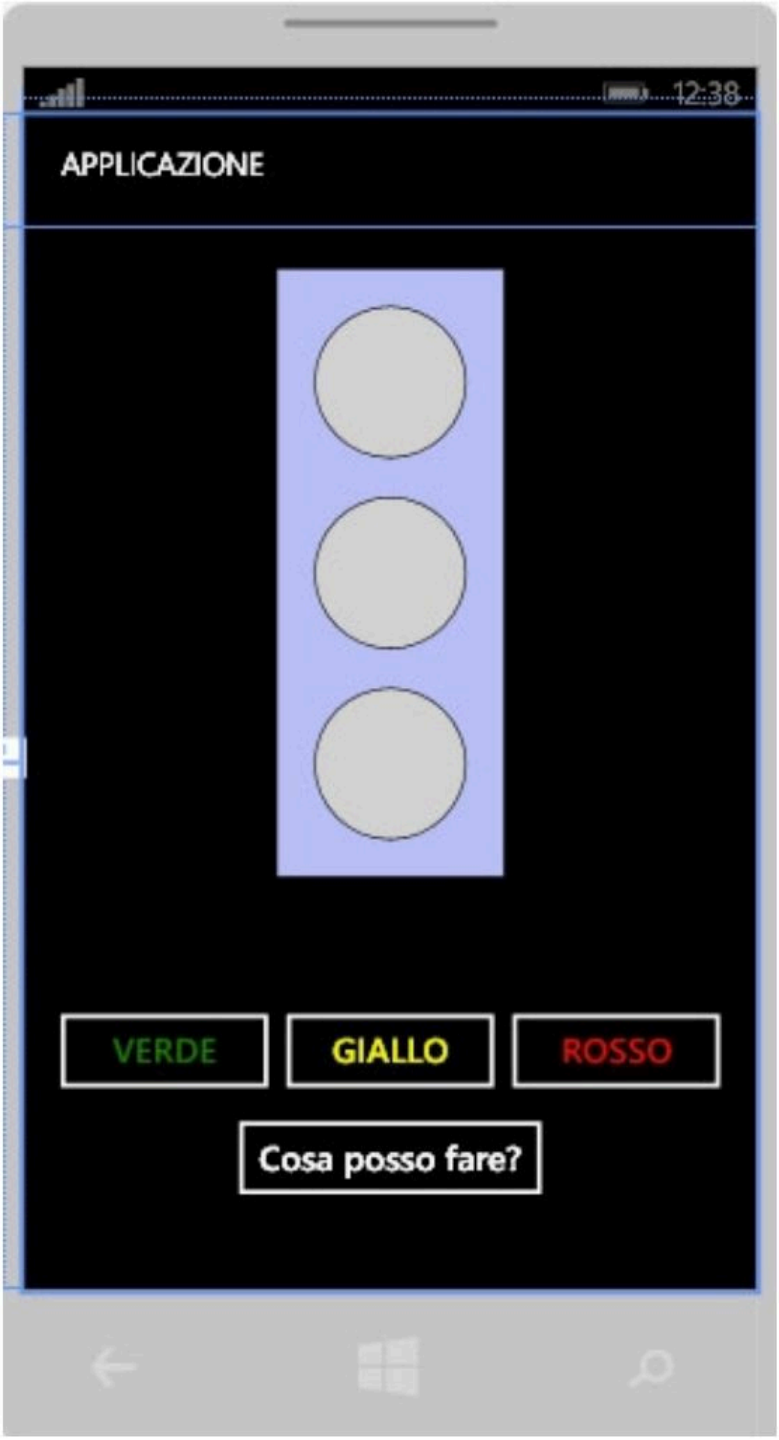
- Content = VERDE;
- HorizontalAlignment = Left;
- Foreground = Green (Verde);
- Margin = 0,0,0,120;
- VerticalAlignment = Bottom;
- Click = button1_Click;

button2:

- Content = GIALLO;
- HorizontalAlignment = Center;
- Foreground = Yellow (Giallo);
- Margin = 0,0,0,120;
- VerticalAlignment = Bottom;
- Click = button2_Click;

button3:

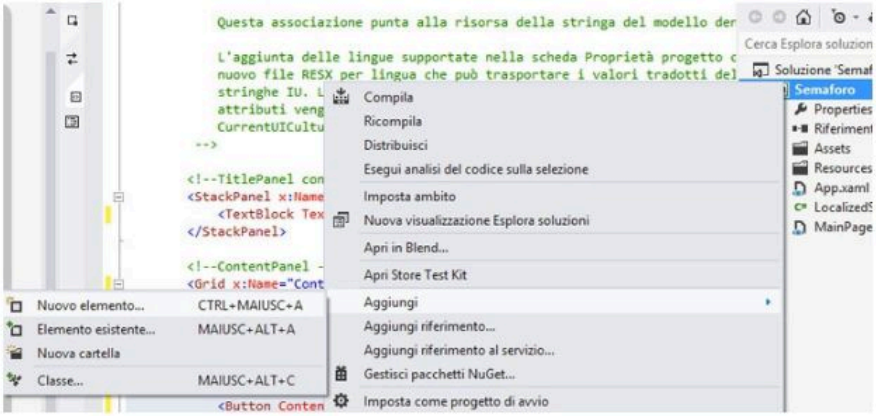
- Content = ROSSO;



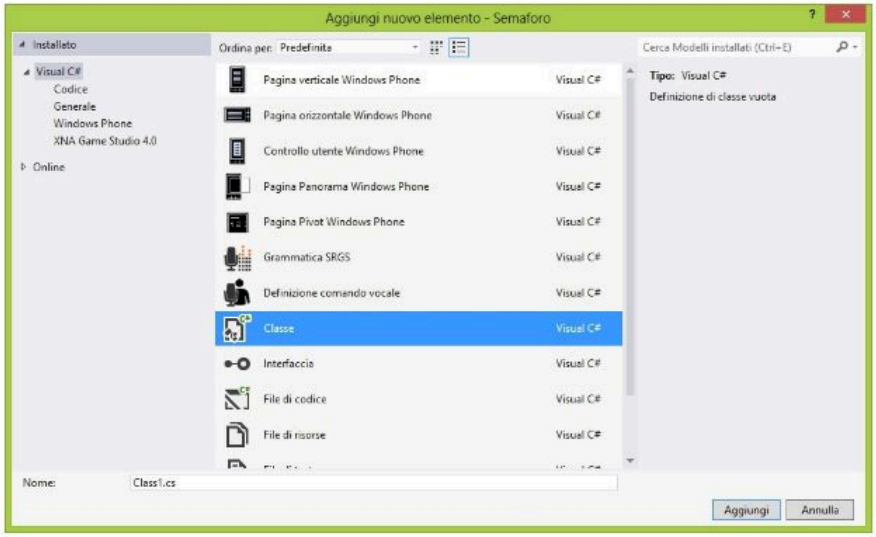
L'interfaccia grafica è pronta, non resta che implementare la parte logica del programma.

5.3 Creazione di un oggetto

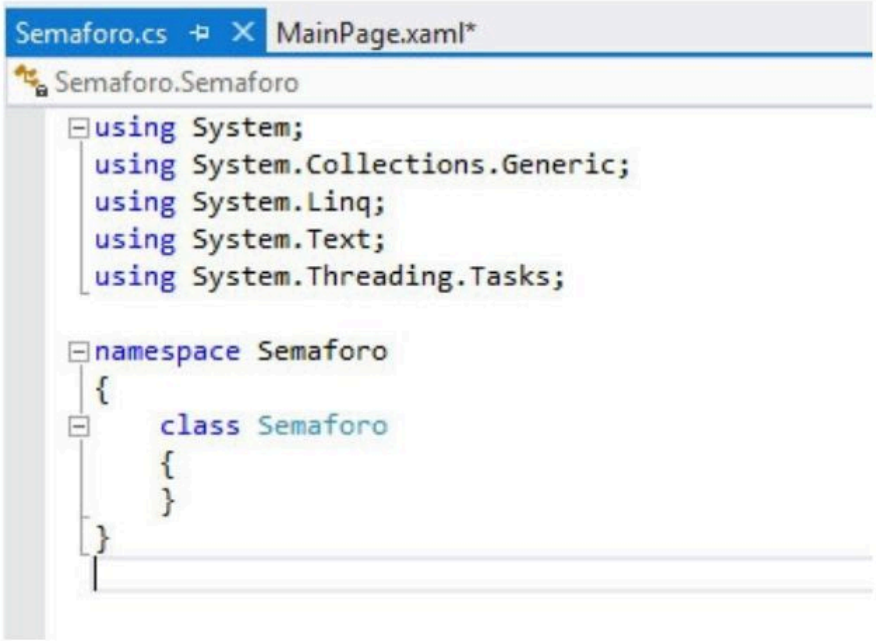
La creazione di un oggetto in Visual Studio è molto semplice. Basta andare su “Esplora soluzioni” e, cliccando col tasto destro sul nome del progetto, selezionare la voce “Aggiungi” e infine “Nuovo elemento...”.



Da qui possiamo visualizzare una lista di scelte. Ai fini di questo progetto, ci servirà creare un nuovo oggetto pertanto selezioniamo “Classe” e nel campo “Nome” inseriamo “Semaforo.cs”.



Notiamo subito nella struttura del progetto l'aggiunta di un nuovo file Semaforo.cs. Questo è molto simile a "MainPage.xaml.cs" e andrà a costituire l'oggetto semaforo della nostra applicazione.



Per prima cosa definiamo le costanti a

cui verrà associato il valore simbolico del colore. Quindi inseriamo il seguente codice all'interno della classe:

```
public class Semaforo
{
    public const int VERDE = 0;
    public const int GIALLO = 1;
    public const int ROSSO = 2;
}
```

Successivamente inseriamo la variabile d'istanza che terrà memoria dello stato del semaforo:

```
private int Stato;
```

Procediamo con la creazione del metodo costruttore, impostando il valore della variabile stato a -1, in quanto il semaforo alla creazione "non è in funzione":

```
public Semaforo()
{
    Stato = -1;
}
```

Adesso definiamo i metodi GetStato e SetStato per consentire a classi esterne di leggere o modificare il valore della variabile Stato:

```
public int GetStato()
{
    return this.Stato;
}

public void SetStato(int
stato)
{
    if (stato == VERDE)
        this.Stato = VERDE;
    else if (stato == GIALLO)
        this.Stato = GIALLO;
    else if (stato == ROSSO)
        this.Stato = ROSSO;
```

```
}
```

Il metodo SetStato deve effettuare un controllo sul parametro passato perchè non tutti i valori sono ammessi.

Per completare la classe definiamo infine il metodo ToString():

```
public string ToString()
{
      switch (Stato)
      {
         case VERDE: return
"Stato semaforo: verde";
         case GIALLO: return
"Stato semaforo: giallo";
         case ROSSO: return
"Stato semaforo: rosso";
         default: return
"Nessuna luce attiva";
      }
}
```

Questo metodo è molto utile in quanto permette all'oggetto di essere

stampato, ovvero di essere espresso come una String (frase). Se questo metodo non fosse definito, verrebbe stampato a video il valore dell'indirizzo di memoria dell'oggetto (es 0XFF123A09).

L'oggetto Semaforo è pronto! Non ci resta che collegare il tutto per completare l'applicazione.

5.4 Completare l'applicazione

Apriamo il file MainPage.xaml.cs.

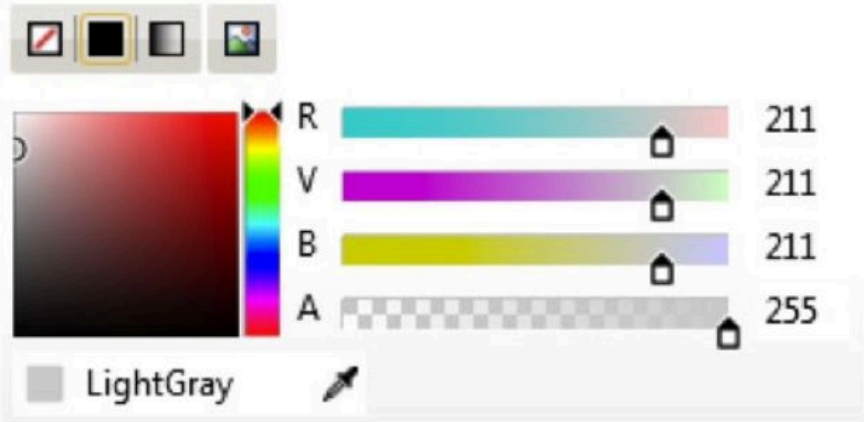
All'inizio del file dobbiamo definire una variabile di tipo Semaforo che servirà per memorizzare lo stato dell'applicazione al verificarsi degli eventi. Inoltre definiremo un nuovo tipo di oggetto "SolidColorBrush" che sarà utile per cambiare il colore dell'Ellipse definita nell'interfaccia grafica. Inseriamo questo codice:

```
// Oggetto Semaforo
private Semaforo semaforo;

// Variabili per definire i colori
private SolidColorBrush inattivo = new SolidColorBrush();
private SolidColorBrush verde = new SolidColorBrush();
private SolidColorBrush giallo = new SolidColorBrush();
private SolidColorBrush rosso = new SolidColorBrush();
```

Una volta create le variabili di istanza, dovremo inizializzarle nel costruttore, quindi inseriamo il seguente codice:

```
// Costruttore
public MainPage()
{
    InitializeComponent();

    // Inizializza l'oggetto semaforo
    semaforo = new Semaforo();

    // Inizializza i colori
    inattivo.Color = Color.FromArgb(255, 211, 211, 211);
    verde.Color = Color.FromArgb(255, 0, 128, 0);
    giallo.Color = Color.FromArgb(255, 255, 255, 0);
    rosso.Color = Color.FromArgb(255, 255, 0, 0);
}
```

Il metodo “FromArgb” della classe Color riceve come parametri il valore ARGB del colore desiderato. Questi valori possono essere visualizzati ad esempio nella casella delle proprietà quando si sceglie un colore.



Come ultima cosa dobbiamo definire i metodi di ogni singolo bottone. I Botton (1), (2), (3), hanno metodi simili e consistono nel settare il valore del semaforo e di conseguenza accendere e spegnere le luci interessate:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    semaforo.SetStato(Semaforo.VERDE);
    ellipse1.Fill = inattivo;
    ellipse2.Fill = inattivo;
    ellipse3.Fill = verde;
}

private void button2_Click(object sender, RoutedEventArgs e)
{
    semaforo.SetStato(Semaforo.GIALLO);
    ellipse1.Fill = inattivo;
    ellipse2.Fill = giallo;
    ellipse3.Fill = inattivo;
}

private void button3_Click(object sender, RoutedEventArgs e)
{
    semaforo.SetStato(Semaforo.ROSSO);
    ellipse1.Fill = rosso;
    ellipse2.Fill = inattivo;
    ellipse3.Fill = inattivo;
}
```

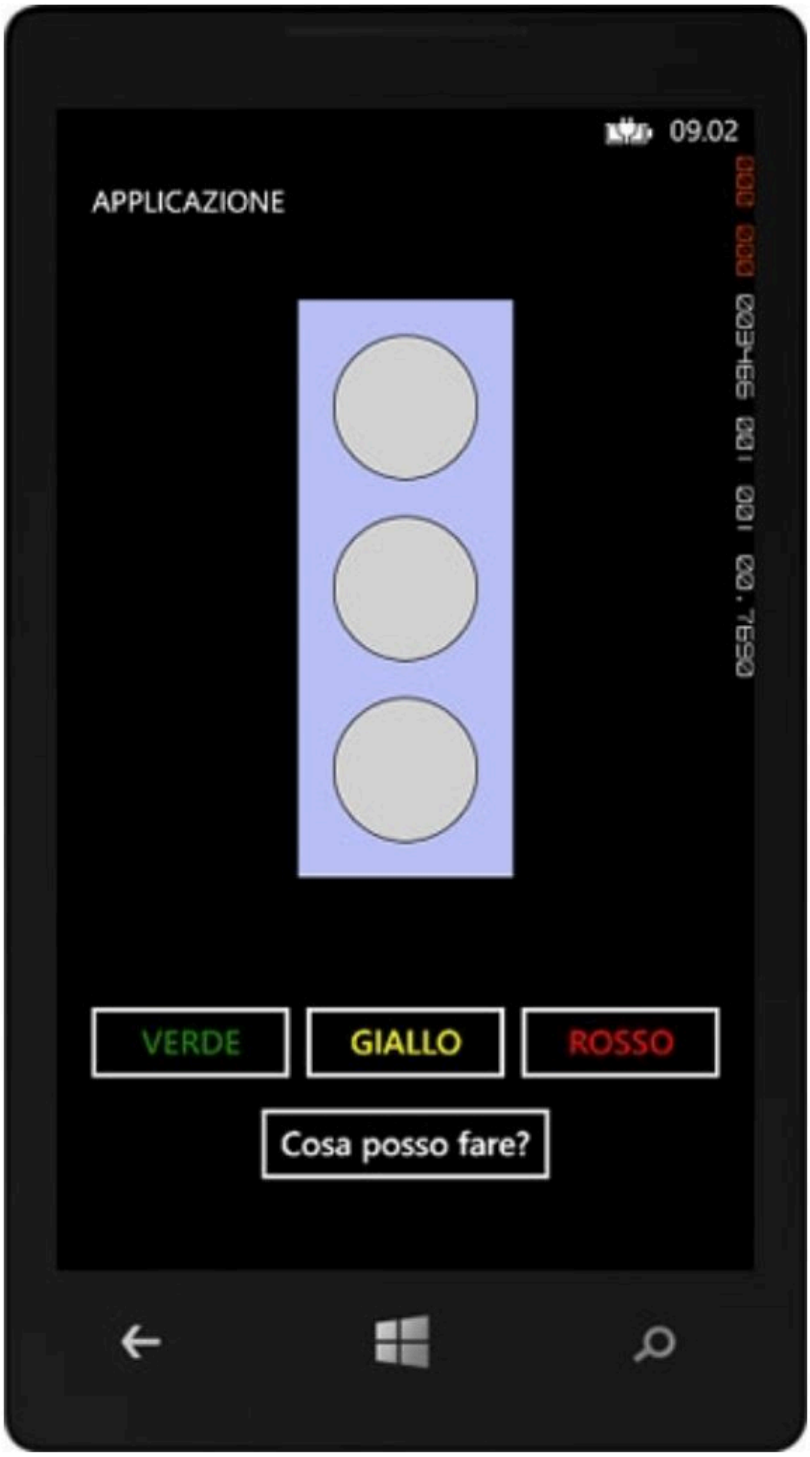
Con ellipse.Fill ci riferiamo al colore dell'Ellipse presente sull'emulatore.

L'ultimo metodo da definire è relativo

al bottone 4. Questo, dopo aver letto lo stato del semaforo, tramite un MessageBox, visualizzerà l'azione possibile.

```
private void button4_Click(object sender, RoutedEventArgs e)
{
    String azione;
    if (semaforo.GetStato() == Semaforo.VERDE)
        azione = "Puoi passare!";
    else if (semaforo.GetStato() == Semaforo.GIALLO)
        azione = "Attenzione! Inizia a rallentare.";
    else if (semaforo.GetStato() == Semaforo.ROSSO)
        azione = "Stop! Non puoi passare.";
    else
        azione = "Ops.. il semaforo è rotto!";
    MessageBox.Show(semaforo.ToString() + "\n" + azione);
}
```

Il progetto è terminato. Non ci resta che provare l'applicazione:

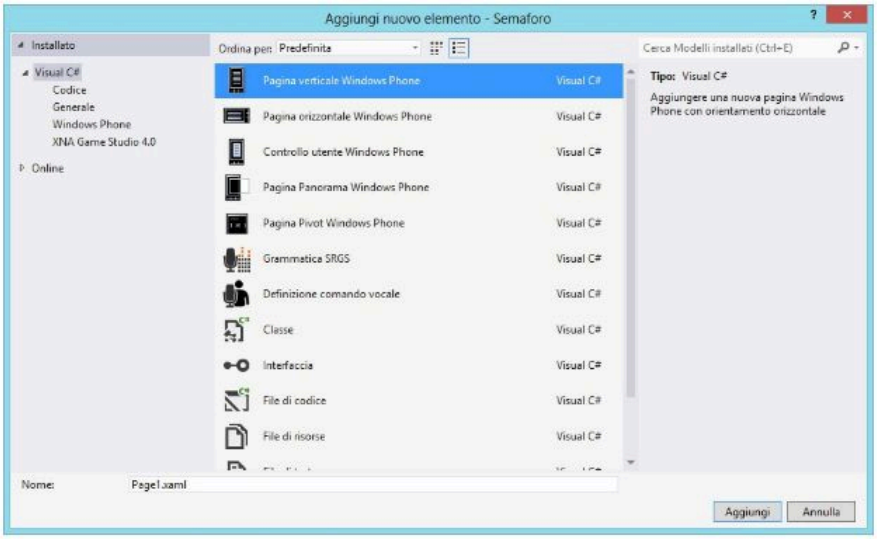

09.02
APPLICAZIONE
VERDE
GIALLO
ROSSO
Cosa posso fare?

09.03
Stato semaforo: verde
Puoi passare!
ok
VERDE
GIALLO
ROSSO
Cosa posso fare?

6. Modificare il semaforo: navigare tra più pagine

6.1 Creare una nuova pagina

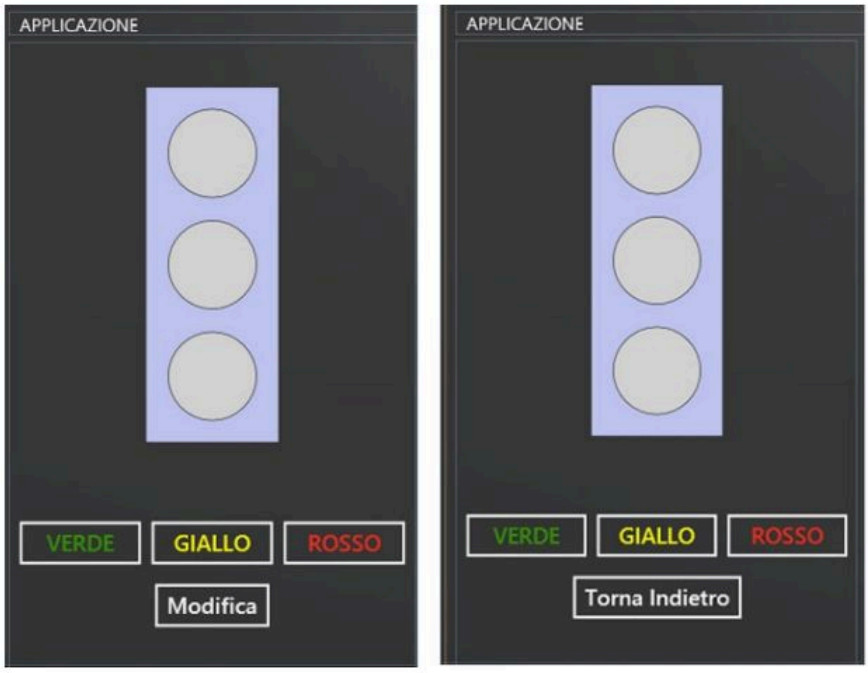
1. Cliccare col tasto destro sul progetto “Semaforo”.
2. Selezionare Aggiungi.
3. Scegliere Nuovo Elemento...
4. Dalla nuova schermata selezionare Pagina Verticale Windows Phone.
5. Cliccare su “Aggiungi”.



Dovremo ora costruire la nuova interfaccia di Page1.xaml esattamente

come con MainPage.xaml ripetendo i passi del capitolo precedente. Per aiutarvi copiate la parte centrale dal file MainPage.xaml come mostrato in figura:

Abbiamo ora a disposizione due pagine con la stessa identica interfaccia. Per distinguerle modifichiamo il testo del bottone “Cosa posso fare?”: nella MainPage inseriamo come attributo Content dell’ultimo Button **Modifica** e in Page1 inseriamo invece **Torna Indietro**.



La nuova pagina è pronta ma, prima di fare qualunque altra cosa, è necessario implementare i metodi relativi ai 4 bottoni che, in questa prima fase, saranno uguali a quelli della pagina principale, per cui trascriviamo l'implementazione dal file MainPage.xaml.cs senza andare a modificare l'intestazione del costruttore.

6.2 Cambio di pagina

Ora modifichiamo il metodo relativo alla pressione del tasto "Modifica" in modo che apra la nuova pagina. A tal

proposito cancelliamo il vecchio contenuto del metodo e aggiungiamo la seguente riga di codice:

```
NavigationService.Navigate(new
Uri("/Page1.xaml",
UriKind.Relative));
```

Il metodo Navigate consente proprio di "navigare" verso la pagina specificata nell'Uri (Uniform Resource Identifier, un indirizzo che rappresenta una pagina in questo caso). Stiamo ordinando all'applicazione di spostarsi alla pagina Page1.xaml. Avviamo l'applicazione e clicchiamo sul pulsante modifica per vedere il risultato.

6.3 Passaggio di dati

Pur modificando lo stato del semaforo nella prima pagina, al momento del cambio, la situazione della nuova pagina non subisce modifiche. Questo succede perché l'oggetto semaforo di Page1 è un oggetto completamente

diverso da quello di MainPage. Affinché le modifiche apportata nella prima pagina possano essere viste anche nella seconda dobbiamo passare il riferimento all'oggetto semaforo originale, per cui modifichiamo il metodo Navigate come segue:

```
NavigationService.Navigate(new
Uri(string.Format("/Page1.xaml
?val1={0}",
semaforo.ToString()),
UriKind.Relative));
```

In questo modo abbiamo aggiunto un parametro al vecchio Uri: la stringa semaforo.ToString() viene sostituita a {0} e viene riconosciuta come parametro val1.

In questo modo abbiamo inviato i dati. Ora dobbiamo fare in modo di riceverli.

Torniamo al file Page1.xaml.cs ed eliminiamo l'inizializzazione dell'oggetto semaforo:

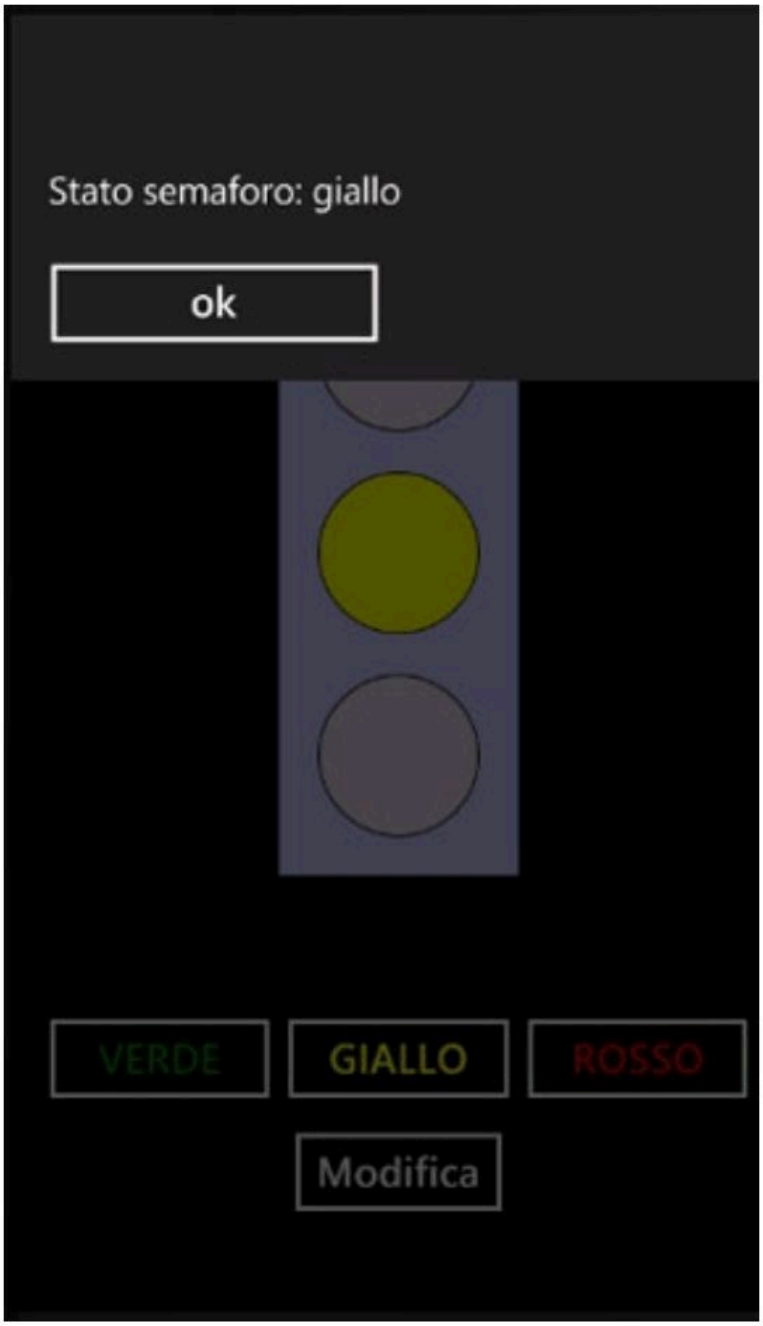
```
public Page1()
{
    InitializeComponent();

    // Inizializza l'oggetto semaforo
    semaforo = new Semaforo();

    // Inizializza i colori
    inattivo.Color = Color.FromArgb(255, 211, 211, 211);
    verde.Color = Color.FromArgb(255, 0, 128, 0);
    giallo.Color = Color.FromArgb(255, 255, 255, 0);
    rosso.Color = Color.FromArgb(255, 255, 0, 0);
}
```

A questo punto creiamo un nuovo metodo come segue:

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    String messaggio = "";
    NavigationContext.QueryString.TryGetValue("val1", out messaggio);

    MessageBox.Show(messaggio);
}
```

Questo metodo viene richiamato quando si naviga verso questa pagina da un'altra e non fa altro che prendere l'oggetto inviato con parametro val1 e mostrarlo sul dispositivo. In questo modo otteniamo un riferimento al ToString() del semaforo originale.

Proviamo l'applicazione per vedere il risultato:



6.4 Variabili globali

Purtroppo i dati che abbiamo passato fin'ora sono di tipo String. Per avere il riferimento al semaforo originale la cosa migliore da fare è utilizzare una variabile globale: una variabile che sia

condivisa da tutte le classi dell'applicazione. Queste variabili vanno dichiarate nel file App.xaml.cs e saranno poi accessibili dall'applicazione stessa. Vediamo come fare:

1. Nel file App.xaml.cs si dichiara la variabile globale semaforo definendo i metodi get e set per la stessa.
2. Nel file MainPage.xaml.cs si sostituisce la variabile semaforo con la variabile app che farà riferimento all'intera applicazione.
3. Bisogna eliminare anche l'inizializzazione del semaforo e inizializzare invece la variabile app:

   ```
   app = Application.Current
   as App;
   ```

4. La variabile da semaforo da inizializzare è adesso quella memorizzata nell'applicazione quindi scriviamo:

   ```
   app.semaforo = new
   ```

```
Semaforo();
```

6. Ora sostituiamo tutti i riferimenti all'oggetto semaforo dei tre bottoni con l'oggetto app.semaforo.
7. A questo punto non serve più passare nulla alla nuova pagina quindi riscriviamo la chiamata al metodo navigate come in precedenza.

```
public partial class App : Application
{
    /// <summary>
    /// Offre facile accesso al frame radice dell'applicazione Windows Phone.
    /// </summary>
    /// <returns>Nome radice dell'applicazione Windows Phone.</returns>
    public static PhoneApplicationFrame RootFrame { get; private set; }

    public Semaforo semaforo { get; set; } 1
```

```
//Oggetto applicazione
App app; 2

// Variabili per definire i colori
private SolidColorBrush inattivo = new SolidColorBrush();
private SolidColorBrush verde = new SolidColorBrush();
private SolidColorBrush giallo = new SolidColorBrush();
private SolidColorBrush rosso = new SolidColorBrush();

// Costruttore
public MainPage()
{
    InitializeComponent();
    //Inizializza la variabile app
    app = Application.Current as App;   3
    //Inizializza la variabile semaforo
    app.semaforo = new Semaforo(); 4
    // Inizializza i colori
    inattivo.Color = Color.FromArgb(255, 211, 211, 211);
    verde.Color = Color.FromArgb(255, 0, 128, 0);
    giallo.Color = Color.FromArgb(255, 255, 255, 0);
    rosso.Color = Color.FromArgb(255, 255, 0, 0);
}

private void button1_Click(object sender, RoutedEventArgs e)
{
    app.semaforo.SetStato(Semaforo.VERDE); 5
    ellipse1.Fill = inattivo;
    ellipse2.Fill = inattivo;
    ellipse3.Fill = verde;
}
```

```
private void button4_Click(object sender, RoutedEventArgs e)
{ 6 this.NavigationService.Navigate(new Uri("/Page1.xaml", UriKind.Relative));
}
```

Ogni cambiamento fatto nella pagina MainPage avrà adesso ripercussioni sulla variabile globale per cui basterà far riferimento a questa variabile anche nel file Page1.xaml.cs:

1. Manteniamo la variabile semaforo in questo caso, ma la inizializziamo con il riferimento

alla variabile globale:

```
App app =
Application.Current as App;

semaforo = app.semaforo;
```

2. A questo punto controlliamo lo stato del semaforo e accendiamo le luci di conseguenza.

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e
{
    base.OnNavigatedTo(e);

1   App app = Application.Current as App;
    semaforo = app.semaforo;

    if (semaforo.GetStato() == Semaforo.VERDE)
    {
        ellipse1.Fill = inattivo;
        ellipse2.Fill = inattivo;
        ellipse3.Fill = verde;
    }
    else if (semaforo.GetStato() == Semaforo.GIALLO)
    {
2       ellipse1.Fill = inattivo;
        ellipse2.Fill = giallo;
        ellipse3.Fill = inattivo;
    }
    else if (semaforo.GetStato() == Semaforo.ROSSO)
    {
        ellipse1.Fill = rosso;
        ellipse2.Fill = inattivo;
        ellipse3.Fill = inattivo;
    }
}
```

Adesso che la variabile semaforo è stata inizializzata con il valore della variabile globale non abbiamo più bisogno di specificare nulla nei tre metodi dei bottoni che resteranno quindi invariati.

Non ci resta che ridefinire il metodo associato al bottone “Torna Indietro” in page1.xaml affinchè ci riporti alla

pagina principale:

```
private void
button4_Click(object sender,
RoutedEventArgs e)
{
      NavigationService.GoBack(
);
}
```

Tornati alla pagina principale bisognerà aggiornare le luci in base ai cambiamenti fatti, quindi inseriamo anche qui il metodo OnNavigateTo con il seguente codice:

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    if (app.semaforo.GetStato() == Semaforo.VERDE)
    {
        ellipse1.Fill = inattivo;
        ellipse2.Fill = inattivo;
        ellipse3.Fill = verde;
    }
    else if (app.semaforo.GetStato() == Semaforo.GIALLO)
    {
        ellipse1.Fill = inattivo;
        ellipse2.Fill = giallo;
        ellipse3.Fill = inattivo;
    }
    else if (app.semaforo.GetStato() == Semaforo.ROSSO)
    {
        ellipse1.Fill = rosso;
        ellipse2.Fill = inattivo;
        ellipse3.Fill = inattivo;
    }
}
```

In questo caso non abbiamo bisogno di prendere nuovamente il riferimento all'oggetto app e poi

all'oggetto semaforo in quanto già memorizzato nella variabile App che non è cambiata.

Parte 3

Approfondimenti

7. Panorama a Pivot

7.1 L'interfaccia Metro

Il look accattivante di Windows Phone 8 non è dovuto solamente alle acclamate Live Tiles, ma soprattutto ai due controlli di navigazione: **Panorama** e **Pivot**.

Panorama e Pivot sono simili, entrambi molto semplici sia da sviluppare che da utilizzare, sono controlli che presentano un modo innovativo di mostrare le informazioni adattandole alle dimensioni ridotte dello schermo e sfruttando le potenzialità del touchscreen.

Le differenze seppur poche sono sostanziali: Pivot divide lo spazio in maniera orizzontale creando dei pannelli che possono essere visti uno alla volta e trascinati verticalmente; Panorama invece ha lo scopo di dare la sensazione di un grande schermo, del quale vediamo però una sola parte

per volta (basta spostarsi verso destra o sinistra per vedere il resto).

7.2 Panorama

Panorama consente di uscire dai confini imposti dalle dimensioni ridotte dei nostri schermi. Finora, infatti, la grandezza dello schermo rappresentava una delle più grandi limitazioni per la programmazione su dispositivi mobili. Con Panorama abbiamo la possibilità di muoverci in ampi spazi con piccoli schermi. L'escamotage è ben riuscito: pur muovendosi tra varie pagine, l'effetto è quello di una sola ampia schermata! In ogni pagina infatti, rappresentata da un PanoramaItem, è visibile la pagina successiva sulla quale ci si può facilmente spostare.

Usare panorama è semplice anche in programmazione, per cui non ci resta che editare il relativo file XAML o più semplicemente aggiungere una pagina con il layout panorama già impostato. Vediamo il primo caso:

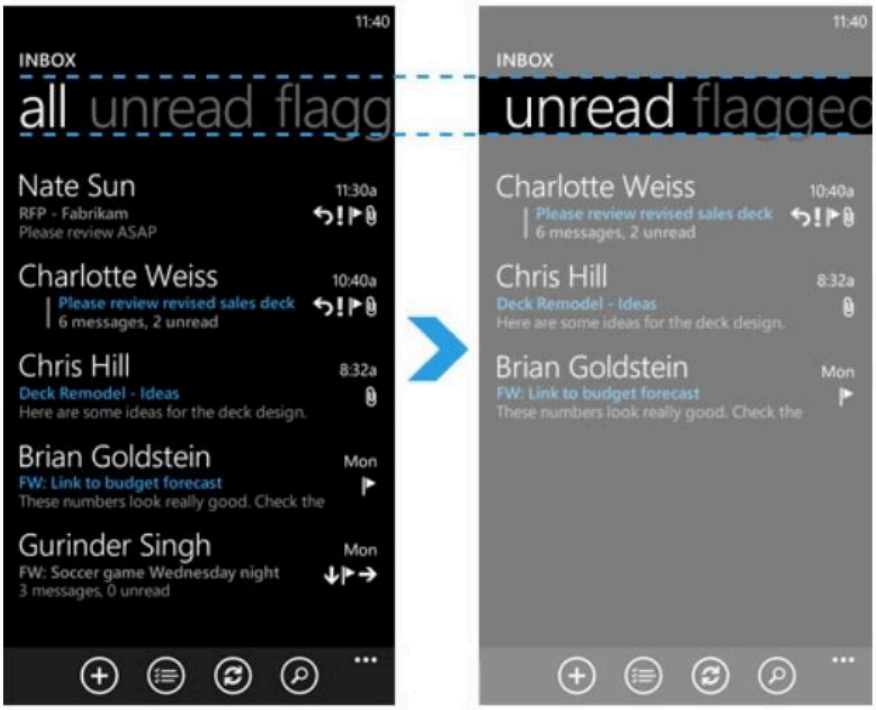
```
<Grid x:Name="LayoutRoot">
    <phone:Panorama Title="applicazione">
        <phone:Panorama.Background>
            <ImageBrush ImageSource="/image_6.png"/>
        </phone:Panorama.Background>

        <!--Elemento Panorama uno-->
        <phone:PanoramaItem Header="item1">
            <Grid>
                <StackPanel>
                    <TextBlock
                Text="Questo è un esempio ..."
                Style="{StaticResource PhoneTextLargeStyle}"
                TextWrapping="Wrap"/>
                </StackPanel>
            </Grid>
        </phone:PanoramaItem>

        <!--Elemento Panorama due-->
        <phone:PanoramaItem Header="item2">
            <Grid/>
        </phone:PanoramaItem>
    </phone:Panorama>
</Grid>
```

Il file XAML è estremamente semplice ed intuitivo, ci siamo cimentati nell'inserire un'immagine di sfondo, un titolo, un blocco di testo come esempio e un nuovo Item del Panorama.

Panorama è una grafica per molteplici utilizzi. Generalmente costituisce la pagina principale dell'applicazione perchè evidenzia tutti i punti salienti di questa senza confondere l'utente/cliente. Non è necessario creare viste cariche di dati e informazioni, grazie a Panorama si può avere una visuale ampia e globale senza perdere di vista le informazioni di maggior interesse.

7.3 Pivot

Il controllo Pivot è rappresentato da una sorta di gestione delle schede ottimizzata, ovviamente, per il touchscreen. Non tutte le schede (che si chiamano PivotItems) vengono mostrate contemporaneamente, esse appaiono all'utente solo se attivate.



Come Panorama, anche Pivot ha uno XAML estremamente semplice e intuitivo. Vediamo subito un esempio:

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <!--Controllo Pivot-->
    <phone:Pivot Title="Esempio Pivot">
        <!--Elemento Pivot uno-->
        <phone:PivotItem Header="item1">
            <Grid>
                <StackPanel Margin="20">
                    <TextBlock
            TextWrapping="Wrap"
            Text="Questo è un esempio ..."
            Style="{StaticResource PhoneTextLargeStyle}">
                    </TextBlock>
                </StackPanel>
            </Grid>
        </phone:PivotItem>

        <!--Elemento Pivot due-->
        <phone:PivotItem Header="item2">
            <Grid/>
        </phone:PivotItem>
    </phone:Pivot>
</Grid>
```

Gli scenari di utilizzo di Pivot sono molti, esso può essere utilizzato per pagine contenenti dati indipendenti tra loro, o che seguono una particolare gerarchia. L'utilizzo classico è mostrare liste di informazioni, talvolta anche dettagliate.

7.4 Modificare la grafica con Visual Studio

Finora abbiamo visto come si manipolano Panorama e Pivot tramite codice XAML. Chi non vuole "sporcarsi le mani" con i vari tag che XAML ci offre, non deve affatto preoccuparsi! Visual Studio ci permette di utilizzare questi controlli senza mettere mano al codice, modificando i parametri che ci interessano tramite la grafica che abbiamo conosciuto.

7.5 Nuovo progetto con Panorama e Pivot

Per creare un'applicazione che contenga un controllo Panorama o Pivot non ci resta che creare un nuovo progetto cliccando su **File > Nuovo Progetto** e scegliere il template di nostro gradimento.

Oppure è possibile semplicemente aggiungere una nuova pagina Panorama/Pivot a un progetto già esistente.

Quindi Panorama e Pivot sono incredibilmente utili per un programmatore, e sarebbe interessante sperimentarli entrambi, date anche le diverse visualizzazioni da adattare al tipo di applicazione creata!

Buon lavoro!

Conclusione

In questo secondo livello del corso abbiamo definito i capi saldi della programmazione orientata agli oggetti, abbiamo imparato a utilizzare le classi, le variabili e i metodi di un oggetto. Abbiamo poi analizzato i parametri e i due principali metodi per il loro passaggio in C# e i controlli della grafica Metro Panorama e Pivot.

Nei nostri progetti pratici abbiamo implementato e gestito figure in 2D e abbiamo definito la nostra applicazione su più pagine tramite il passaggio di stringhe e l'utilizzo delle variabili globali.

Nel prossimo livello oltre a imparare nuovi costrutti C# e strumenti per la POO, creeremo applicazioni più complesse sfruttando nuovi metodi e saremo capaci di realizzare, ad esempio, una galleria di immagini per la nostra app.

Esperto in un click

Windows Phone 8
IMPARERAI
Windows Phone 8
applica
Windows Phone 8
Windows Phone 8
Mio Browser
area51
Windows Phone 8
Impostazioni

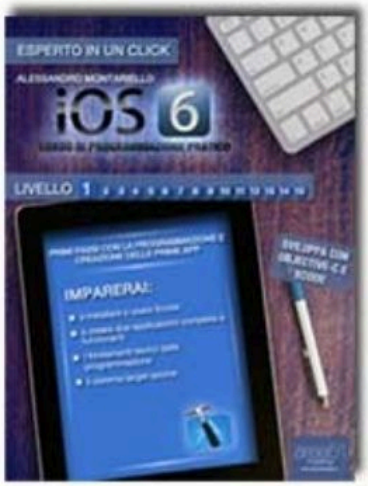
enrico de benedictis
ePUB
manuale pratico
Chapter III
Chapter IV
area51
Publishing

ESPERTO IN UN CLICK
LINUX
PRIMI PASSI CON LINUX
IMPARERAI:
ESPERTO IN UN CLICK
LINUX
LE APPLICAZIONI PER INTERNET E PER L'UFFICIO
IMPARERAI:
ESPERTO IN UN CLICK
LINUX
LE APPLICAZIONI MULTIMEDIALI
IMPARERAI:
ESPERTO IN UN CLICK
LINUX
LINUX SUI SERVER
IMPARERAI:
ESPERTO IN UN CLICK
LINUX
LE APPLICAZIONI PER SERVER
IMPARERAI:

ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI A:
+ iOS
ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS
ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS
ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS
ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS
ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS

LAMP
Guida per creare il tuo sito
IMPARERAI:
LAMP
Guida per creare il tuo sito
IMPARERAI:
LAMP
Guida per creare il tuo sito
IMPARERAI:

WEB MARKETING
Francesco De Nobili
SEO Google
Guida al web marketing
con gli strumenti di Google
SEO
ADWORDS
TRENDS
SUGGEST
AD PLANNER
ANALYTICS
AUTORSHIP
GOOGLE PLUS
FACEBOOK
TWITTER
YOUTUBE
area51
Publishing


ESPERTO IN UN CLICK
iOS 6
LIVELLO 1
IMPARERAI:

ESPERTO IN UN CLICK
iOS 6
LIVELLO 2
IMPARERAI:

ESPERTO IN UN CLICK
iOS 6
LIVELLO 3
IMPARERAI:

ESPERTO IN UN CLICK
iOS 6
LIVELLO 4
IMPARERAI: